Using XP in a Big Process Company
A Report From the Field

James Grenning
Object Mentor, Incorporated
565 Lakeview Parkway
Suite 135
Vernon Hills, IL 60061
(847) 573-1565
grenning@objectmentor.com

ABSTRACT
The paper describes a project developed using many XP practices in a company that has traditional formal process. The paper covers how XP was proposed to the management, how the project seed was started and grown, and some of the issues the team faced during its first six months.

Keywords

XP, Extreme Programming, software process, process change, Object-Oriented Design, software project management.

1 INTRODUCTION
This is a story about adapting XP in a company with a large formal software development process. The company, let's call it Defined Process Systems (DPS), is involved in safety critical systems development. DPS is developing a new system to replace an existing legacy system. The system has been divided into subsystems through a systems engineering effort. I was brought in to help one of the teams start their project using iterative development, use cases and Object Oriented Design (OOD) techniques. The project is essentially an embedded systems application, running on Windows NT and is part of a network of other machines that must collaborate to provide their services.

Shortly after starting this project I spent a week at a training class called XP Immersion™ I. XP Immersion is a one-week intensive training class in the techniques and philosophy of Extreme Programming[i]. Enthused by XP, I talked to the DPS team about applying some of the XP practices to our piece of the project. XP was very different than the standard process at DPS as is use iterative use case-driven object-oriented development. So, why not press our luck a little further. DPS decided to try iterative development and Object Oriented techniques. They were game for something different, but how different? We decided to take the idea of applying XP practices to the Director of Engineering.

2 CURRENT SITUATION
The standard practice in the division was built in response to many problems. For one, most of the developers were light on experience. The more senior developers had the role of reviewing and approving the work of the junior developers. To their credit, a formal review process was in operation and taken very seriously by the group. They were good at it. Many steps were automated. Issues and defects were captured on the web. Review meetings were crisp.

In the world of Big Design Up Front (BDUF) and phase containment these guys were good. I could end this article right now except for one problem. All this process adds overhead to the development of software. To design something I need to figure out the design, document it in Rose, schedule a review meeting, and distribute materials to review. Then reviewers have to review the materials and enter issues on the web, the review meeting is held, issues and defects have to be closed out on the web, documents have to be fixed and desk checked. I am getting winded just thinking of all this.

All this up front work was not keeping bugs out of the integration and deployment stages either. Engineers often were getting burned out and heading for systems engineering or management. These were engineers that were just getting their skills to a decent technical depth. Unrealistic deadlines and surprises late in the project were taking their toll.

The team was also struggling with how to get the project started. Requirements were in a prose and only fully understood by the person that wrote them.

To summarize the problems needing to be addressed: the process has a lot of overhead, deadlines were tight, engineers were running away, requirements were partially defined, and they had to get a project started. With all these issues am I going to go in and propose something as extreme as XP? I believed it would help so I did.

3 CHOOSING YOUR BATTLES
I just finished telling you that the company is a BDUF shop, with lots of paper, web pages, reviews and approvals. Just write the product features on note cards, don't do any up front design and start coding… are you nuts! To people unfamiliar with XP this sounds a lot like hacking. How did we get by this natural objection?

Having socialized XP with the group, we understood what the main objections would be as we tried to sell XP to the management team. Like a good lawyer, we prepared some questions, along with the expected answers for our presentation. We expected that the decision makers would think that some of the practices were dangerous and could not work at DPS. The need for documentation is ingrained in the culture so we expected they would be concerned that we were not going to do documentation. Could the code be the design? Can we really build a product without the up front design? What if there is thrashing while refactoring? What about design reviews?

We decided to choose our battles. To paraphrase Kent, *do all of XP before trying to customize it*. I think that is great advice, but for this environment we would never have gotten the OK to mark up the first index card. We needed to get some of the beneficial practices into the project and not get hurt by leaving other practices behind. We did not leave behind practices that we didn't feel like doing; we tried to do as many as we could. We used the practices and their interactions as ways to sell around the objections.

We started by identifying the goals of the project. We need a working product with reliable operation, with enough documentation to allow effective maintenance (no more, no less), and with understandable source code. This is as objectionable as motherhood and apple pie. The *standard process* would identify the same objectives.

We all agreed that a reliable working product is a critical output of the project. This is particularly important, as this is a safety critical system. A tougher question is what is enough documentation? This is where it gets interesting. This application is not your typical XP target application. It is part of a larger system, being developed by multiple groups at multiple sites. These other groups are not using XP or short iteration development. They are using the standard DPS process. We have a potential impedance mismatch between the XP team and the rest of the project.

Proposing no documentation would end the conversation. Let's keep the conversation going and answer a question with a question. What do we want from our documentation? There has to be *enough* documentation to support the maintainers of the system. There has to be clean and understandable source code. Due to the impedance mismatch between groups, some form of interface documentation is needed. There has to be some documentation to define the *requirements* of the product. We need some documentation for technical reviews. These answers do not all align with XP out of the book, but it kept the conversation going. Keep in mind that XP is not anti-documentation. XP recognizes that documentation has a cost and it may be more cost effective to not do the documentation. This of course, violates conventional wisdom.

After acknowledging and addressing the objectives of the project, I led the team through the cost of change pitch from Kent Beck's book Extreme Programming Explained[ii]. The director, the manager, and some senior technologists agreed XP addresses many of their current development problems. They also thought XP right out of the book would not work for them. What did we want to do differently?

Documentation and reviews are going to be the big roadblocks. So let's go through each objection and see the choices we made.

Objection: "Requirements" on note cards! "I can't give a stack of note cards to the test team.", "Bob in firmware needs the cards too.", "Someone will loose the cards.". I noticed that the "standard process" allowed use cases in the form described by Alistair Cockburn[iii]. This is a text-based method, which is like user stories with more details. We also had other groups that needed to look at the use cases, so we decided not to fight that battle. We had enough battles lined up already. Use cases it was.

Objection: "We need documentation for the future generations of engineers that will maintain the product we are building. We need our senior people to look at the design to make sure you guys don't royally screw this up."

The objections were answered like this:

The best way to give a good future to our software maintenance decedents is to provide them with some good clean and simple source code, not binders full of out-of-date paper. Maintainers always go to the source code, it cannot lie. So, let's give them really good source code with a little documentation so they can figure out how to get around in the system. XP relies on the source code being simple and expressive. Refactoring is done to keep it that way. The source code is the design!

A follow-on objection is that what one person thinks is readable source code is not to another. XP addresses this through pair programming. If a pair works hard at making the source readable, there is a really good chance that the third programmer to see the code will find it readable too. Plus, with collective code ownership, anyone can change the source code if need be.

Another follow-on objection is that code is not enough. When a project is mothballed, the maintainers will need more than source code. Some documentation or transition strategy is needed when a project is put on the shelf or transferred to another team. The best documentation that could be given to the maintainers is the documentation that describes the state of the software at the time it is put on the shelf.

This document can and should also be written to avoid needing to be changed. As maintenance proceeds, the document will undoubtedly be neglected. Keep the details

out of the documentation. Make the document high level enough to not be affected by the usual maintenance changes and bug fixes. Let the documentation guide the future developer to the right part of the code, then the high-quality, readable, simple source code can be used to work out the details.

Following this strategy will not result in a huge document. Remember you have some of the best detailed documentation available in the form of automated unit tests; working code examples of exactly how to use each object in the system. Documentation is not prohibited in XP. Realize it has a cost and make sure it is worth it. Documentation tasks can be planned into any iteration. The advice here is to document as built not as anticipated.

The next follow-on objection is that we'll never do the document at the end of the project. So instead you want to do a little bit at a time, and have to keep changing it and rewriting it? Doesn't this sound like it will take a lot time? It does! So the management team must stick to its guns and do the high level documentation task at the end of the project. Pay for it with less time wasted during the development.

They did not completely buy this one and hence the final objection: "I still don't believe you. What if the design is no good? Do I have to wait until the end of the project to find out?" Pair programming was not reason enough for the management team to give up their review process. The big problem with their current review process is that it guarantees that development will go slowly. The process goes something like this: create a design, document it in rose, schedule a review meeting, distribute the materials, have everyone reviews the materials, have the review meeting, collect the issues, fix the issues, maybe do another review, then finally write some code to see if the design works. This makes sure the cost of change is high. There is also considerable dead time in which meaningful progress on the project could be made. Because of the safety critical nature of the application, the management team was not willing to give up on these reviews.

I proposed a more efficient way to do the reviews: the ask-for-forgiveness vs. ask-for-permission design process. Let the development-team work for a month at a time on the system. At the end of the month, a design-as-built review package would be assembled and reviewed. This takes the review off the critical path, so the review process would not slow down the team. We agreed to document the significant designs within the iteration and review them with the review team. Issues found by the reviewers would be put in as stories to the next iteration. The idea here is to spend a small amount of time in the iteration documenting the design decisions that month. As it turned out, we really did not have to ask for forgiveness at all.

4 IT'S NOT ABOUT XP

It's about building better software predictably and faster.

We compromised on a number of issues but unmentioned is that we had agreement to use many of the XP practices: test first programming, pair programming, short iteration, continuous integration, Refactoring, planning, and customer team member. We added some process and formality: use cases, monthly design reviews, and some documentation. This was a significant change in how software was developed and we were hoping to prove it is an improvement. The director still had his concerns. He thought XP offered a lot of promise for a better way to work that could lead to improved quality, faster development, better predictability, and more on-the-job satisfaction ("smiles per hour"). He said we are "making footprints in the sand.". He could see that this could really change how software is developed. If we could show one of the potential benefits and don't make the others worse he though it might be worth doing XP. If we can impact two or more of the factors there is a really big payoff.

The problems of poor quality, delays in delivery, long delivery cycles, and burned-out engineers plague the software industry. XP struck a cord with the leaders of the team that was forming. The techniques appeared to address some of the problems the team was facing. The focus on testing and pair programming could support the team's effort to build a quality product. The iterative nature of XP could help the team determine how fast it could go and give the needed feedback to the management team.

So, it's about getting the job done. XP was just a set of techniques that seemed very promising

5 EXPLORATION — JUMP START

Many projects get stalled in the "fuzzy front end"[iv]. This is especially a problem in waterfall methodology projects. All the requirements have to be defined prior to starting the design process. In XP as soon as you know a couple weeks worth of user needs you can start development. Think of how hard it is to shave a month off the end of a project. Think of how easy it could be to save a month on a project just by starting development as soon as a month worth of stories were identified. Story development occurs concurrently with story implementation.

The customer (the systems engineer) on the project had a requirements document written. We took the document and read each paragraph. As we identified a functional requirement we wrote it on a card. This process is called use case identification. We did not bother to elaborate the use cases, just name them. Over a few days we identified 125 use cases. With this list it was pretty easy to pick out the most important use cases.

In XP the most valuable user stories would be chosen and discussed with the programmers. We were using use cases, but the same idea works fine. The customer chose the most

valuable use cases and elaborated them. We decided not to worry about the use case extensions[v] (ala Alistair Cockburn) in the first iterations. Use case extensions per Alistair's method hold the special cases or variations from the nominal flow of the use case. Because we were using XP, we believed that we could ignore the details and not be penalized later. In the early iterations we kept the product definition simple. We assumed that there are no special cases or error cases. We also did not bother using use case diagrams, as they did not add any value to the development team. At the beginning of a project you need to believe that the design can and will evolve. Otherwise the desire to do up front spec work will put the team into analysis paralysis. Knowing that you can evolve the design sets you free to start building the system as soon as some functionality is identified.

6 FIRST ITERATION

In the first iteration there were three people on the team, a customer and two developers, me included. We started by getting CppUnit set up and integrated with our development environment, VC++. This did not take long. The tools are pretty easy to use. Our main goal in the first iteration was to build a little bit of the product, get some experience, and build some skill and confidence.

Our customer provided us with the most important use cases. He selected the main line features, and did not select the parts of the use cases that included the special cases (use case extensions). It really helps to make these simplifying and scope limiting decisions each iteration. The core features drive the design. Simplifying assumptions keep complexity out of the code, at least temporarily. To quote Grady Booch[vi] quoting J Gall: "A complex system that works is invariably found to have evolved from a simple systems that works."[vii]

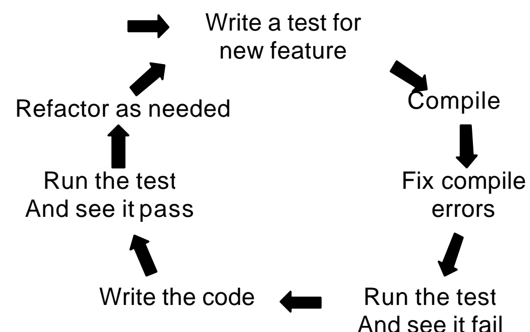
We took the features included in the first iteration and had our planning meeting. My partner and I had OOD experience but no real XP experience (except for the week I spent at XP Immersion™ I). We wanted a guide to our first iteration so we spent about half a day with a whiteboard looking at design ideas. Because we were unsure of XP, we were not sure if we could really start coding without doing some design. So we did a little bit of design or a Little Design Up Front (LDUF). We found a group of collaborating objects we thought would be meet the needs of our first stories, copied the whiteboard class diagram and sequence diagram, and went to work. We did not put the design in Rose. We worked from a hand drawn copy. We knew things would change and we did not want to waste our time. It helped us get organized and gave us a vision of where we were going.

The system we were building had custom hardware, and conveniently enough the hardware was not available. It was still in development. So how are we going to write code for hardware that does not exist? We identify and evolve an interface to the hardware and simulate its behavior. It is actually a luxury in disguise to not have the hardware! It keeps real hardware dependencies from creeping into the code. How could I get hardware dependencies in my application logic if I don't have any hardware?

If the real hardware existed, I could write code that would use the hardware API directly. If it doesn't I can't. Why should I care? Flexible, clean software is critical to being able to do XP. The more dependency my software has on its execution environment the harder it is to test it independently. It is less flexible if it has dependency on the hardware API. The APIs will warp the application. Also, hardware has this nasty habit of changing (parts obsolescence, cost reductions, new technology, etc.). I don't want those changes to ripple through my application.

Not having the hardware means that I must abstract it. I can let the application tell me, through evolution, what the interaction with the hardware needs to be. One of my OO mantras is "the interface exists for the benefit of the client."[viii]. Let the client define what the hardware interactions need to be, and provide an interface that provides those features. A nice side-affect is that the client code can be tested independently from the hardware. Independence is key to being able to build effective unit test cases. This may sound like BDUF. We did not think it was because we did not try to nail down all the interface particulars. Just the place holder with a couple ideas of what the methods might be. When using C++, interfaces are created using an abstract base class (a class with all pure virtual methods). In Java, an *interface* is used. This dependency discussion could be repeated for isolating other entities such as the database schema, GUI and protocol. For more information see Robert Martins paper titled *Principles and Patterns*[ix].

During our iteration planning meeting and our LDUF session, we identified some of the interfaces needed to support the Iteration 1 features. These act as placeholders for the real hardware. We add simulations behind the interfaces. Typically these interfaces provide some canned data or write their output to a file. Add an example.



So we sat down to write our first line of code. We had our LDUF hand drawn diagram. We picked a candidate class from the page. We followed the test first process. Our first line of code was a test. The test did not compile. We fixed the compile. We ran the test. It failed. We fixed the test. We are finally in maintenance!

We established pair programming guidelines. We decided that test cases, simulators and interface classes could be developed solo, but all other production code must be done in pairs. During the first iteration we only were one pair, which meant some significant downtime. Meetings were a real productivity killer. We found pair programming to be fun and intense. We found when working in pairs we stayed on task. If one partner lost track of where we were going, the other partner quickly re-synched their efforts. We taught each other about the tools and learned new skills.

My partner and I had a compatible coding style. If we did not we would have had to negotiate a coding style. Our goal is to have the code look like it was not written by a lot of different people. Why you ask? The code needs to be easy to read. We did not want a big coding standard (there was one we generally ignored). We just wanted the code to read the same. The style really does not matter, so we would adapt. Later, as people joined the team, a coding standard was established and was self-documenting. The basic coding standard is this: “Make the new or modified code look like the code that is already there.” Cool, a one line coding standard! There was pressure to use the department coding standard. From a “choosing your battles” point of view, we gave into using the standard comment blocks in front of each function. These generally did not provide much information or value. The class level comments were better. They described the purpose of the class. For the most part the standard commenting practice brought a lot of noise into the code and distracted the programmers.

7 MORE ITERATIONS

Considering we made a lot of simplifying assumptions in the first iteration, we wanted to know how the system would handle bringing in some of the special cases. Would refactoring work as advertised? Could we keep the design clean and add this functionality? We found that bringing in new more complex use cases did flow well. We got the functionality working, and then refactored the code to a clean state. On a few occasions, we chose to refactor the code so the new feature looked like a natural fit, then just dropped in the new feature. This was right out of the book!

One other significant simplifying assumption was that the system would only have to support a single user transaction at a time. Would the design we started be able to handle 50 or 100 concurrent transactions? Could it scale? We decided to bring in the Active Object design pattern[x], so we could keep our threading policies separate from our application logic. Active Object is basically an object that encapsulates a thread. The thread can delegate to the application object in a queued manner. This allowed us to practice separation of concerns. Separate the application logic from the threading model. We built up a bit of initialization to create the fifty instances of the system core needed to support the fifty users. Then we pumped in a traffic simulation of a worst-case scenario.

The test ran fine. The performance of the system was measured and we determined that about 5% of the CPU was used to support a greater than worst-case load. Granted, all the IO was stubbed out and simulated. We were able to show early in the development that one of our greatest risks, not being able to handle the worst case load, would not be caused by the threading model we chose.

8 EVOLUTIONARY DESIGN

We planted a seed of functionality at the center of this subsystem and simulated its interactions with its environment. New stories are brought in that make the seed grow and incrementally bring in more functionality and complexity. We have been able to watch the design grow and evolve. The core design has not changed much since the first iteration, but the system has grown. From its simple beginning of a half a dozen classes and a few simulations, the design has evolved to 100 classes. This took about six months with a small team. It’s not a huge system but DPS would not have produced as clean and flexible a design as quickly and with such confidence.

Evolutionary design relieves a lot of pressure from the team. We don’t have to create the best design of all time for things we are not quite sure about. We only need to create the best design for what we know about today. We made good design decisions one at a time. Naïve design decisions did not cost us much if anything and supported our bias toward action. Our automated tests and refactoring give us the confidence that we can continue to evolve the system. It is not to say we can turn an automated teller machine into a payroll system by refactoring. We’re assuming there is at least a building of features on top of the existing features.

Object Oriented Design (OOD) is not required to practice XP, but it really helps. The test first programming technique promotes building and testing software in small independent pieces. Small independent pieces are really hard to create in procedural languages like C. Independent pieces are much easier to develop using OO programming languages such as Java, C++ or Smalltalk. Independent classes talk to other classes through their interface. Interfaces provide a flexibility or substitutability point. These are exploited by plugging in stubs of the real application classes allowing independent testing. These interfaces protect one part of the software from knowledge of other parts of the software. See Principles and Patterns paper referenced earlier.

9 PROJECT MANAGER SURPRISES

Not only were the programmers happy with their creation, after the 4th iteration the project manager said:

**"I'd only have three documents by now!"
"Instead I have a piece
of the system that works!"**

The project manager discovered another benefit. Usually a project manager coordinating the work of the team with other teams spends a lot of time juggling priorities and figuring out task dependencies. On the XP team, dependencies between features were almost non-existent. Features are done in the order of customer priority, not internal software framework order dictated by the BDUF. The team doing XP was agile and able to adapt to the changing needs of the other sub-systems.

10 BUILDING THE TEAM

Building the team, while skill is being developed, was done slowly. We felt we could absorb one or two people per iteration. We did not let them take tasks right away. Newcomers were mainly used as pair partners during the first iteration. Then as they got to know the system and our practices, they started to take tasks at the iteration planning meetings. The team velocity would not be assumed to go up when a new person was added to the team. The team velocity is measured, not predicted.

The DPS way of developing software made up for programmer experience by having senior engineers review the work of the of the less experienced engineers. In XP projects there is still a spread of expertise that has to be addressed. It becomes very critical to have at least one senior engineer on the team. We don't go giving them a big title, or a special role, but senior people are needed. They help spread the wealth of knowledge. They pair with less experienced engineers. They both learn.

11 SPREADING PRACTICES TO OTHER GROUPS

You can bring a horse to water, but you can't make him drink. What I have seen is that when the engineers want XP the management does not. If management wants XP the engineers don't.

To the managers: The thing that works the best is to have open-minded engineers and the OK to try new things. Encourage the practices but don't mandate them. Give them some good coaching. Challenge them to go against the status quo. Recruit a team that wants to try XP rather than force a team to use XP. Make sure the organization sees that no one will be shot for trying something different. Forcing XP on team members can alienate them. This causes resistance. Get some coaching help. When trying some of the XP practices, but not all, problems may surface. The coach can identify problems and show the team how the practices may reduce the problem. Try a practice for an iteration or two. If it does not work we can change. Iterations are short. Feedback comes often.

To the engineers: try a sales pitch like the one above. Tell the management that we can try XP for a while and if it does not work, we can change. Iterations are short. Feedback comes often.

12 REFLECTION

Open workspace is an often-overlooked practice and we overlooked it. Cubes were spread out and the workstations were in the corners. This makes pair programming and collaboration really hard. Bite the bullet and build an open workspace.

Automated acceptance testing is needed. Ideally the customer would write test scripts in a high-level application specific language. We have the scripting language, but the engineers write most of the tests. I would like to see less work done documenting use case steps as an input to development and more effort spent on creating test scripts. These test scripts provide the rigor that is missing from the user story. If you think of it, writing the use cases steps and extensions and then writing test scripts from them is essentially doubling the work.

Short iterations (two weeks) give rapid feedback. Two weeks seems really short if you have not done XP before. We chose a four-week iteration length. It works better than two or three-month iterations but two-week iterations give much more feedback. Iteration length creep happened frequently.

The team does not miss the up front design. They are very happy with the flexibility of the design. The document created for the review process is just a document created for the review process. It is used to do a periodic review and is not used much by the team. I would encourage the team to challenge the need for the periodic review. Does it help the team? How would the time be better spent?

13 INFORMATION AND QUESTIONS

For more information, contact:
grenning@objectmentor.com.

ACKNOWLEDGEMENTS

I would like to thank the real client that provided the experience to write this paper (who wished to remain anonymous). I also want to thank Chris Biegay and Jennifer Kohnke for a job well done in helping me prepare this paper.

REFERENCES

[i] Kent Beck, Extreme Programming Explained First Edition. Addison-Wesley Publishing, Co. (October, 1999)

[ii] ibid

[iii] Cockburn, Alistair, Writing Effective Use Cases (The Crystal Collection for Software Professionals) First Edition. Addison-Wesley Publishing, Co. (October, 2000)

[iv] Steve McConnell, Rapid Development First Edition. Microsoft Press. (July, 1996)

[v] Alistair Cockburn

[vi] Booch, Grady. Object-Oriented Analysis and Design with Applications. Second Edition. Santa Clara, CA. The Benjamin/Cummings Publishing Company, Inc. Second Edition. Addison-Wesley Publishing, Co. (February, 1994)

[vii] Gall, J. 1986, Systemantics: How Systems Really Work and How They Fail. Second edition. Ann Arbor, MI: The General Systemantics Press (1977)

[viii] Robert Martin, from *Principles of Object-Oriented Analysis and Design* class

[ix] Robert Martin, Principles and Patterns, www.objectmentor.com

[x] Schmidt, Douglas C. and Lavender, R Greg. *Active Object, http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf*